An Empirical Study Of Commonly Used Modeling Techniques For Software Fault Prediction

Prepared for

NASA Independent Verification and Validation Facility

FAU Technical Report TR-CSE-01-32

Taghi M. Khoshgoftaar*
Naeem Seliya
Florida Atlantic University
Boca Raton, Florida USA

July 2001

*Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Deptartment of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel/.

Executive Summary

High-assurance and complex mission-critical software systems are heavily dependent on reliability of their underlying software applications. Predicting faults early in the software life cycle can be used to improve software process control and achieve high software reliability. Timely predictions of faults in software modules can be used to direct cost-effective quality enhancement efforts to modules that are likely to have a high number of faults. Prediction models based on software metrics, can estimate number of faults in software modules. Software metrics are attributes of the software system and may include process, product, and execution metrics.

A brief overview of a few software quality prediction modeling techniques is presented in [15], as part of our research study sponsored by NASA's Independent Verification and Validation Facility [17]. Software quality models can be used to predict quantitative values of a quality factor for software modules, e.x., number of faults. Software organizations are interested in knowing the most optimum way of utilizing their available quality improvement resources. Prediction models that estimate number of faults in modules can be effective tools in software quality estimation problems. An effective approach that further enhances the usefulness of fault prediction models, is to allocate quality improvement resources by prioritizing high-risk modules over low-risk modules.

Module-Order Modeling (MOM) is an effective approach that facilitates ranking of modules with respect to their predicted risk factor (for example number of faults). Such a ranking can then be used by project managers to allocate enhancement activities to needed areas depending on resource availability. Proposed continuation (NASA Center Initiative FY2002) of our ongoing research with software quality estimation models [17] will involve extensive research with module-order models.

An inherent feature of a module-order model is the underlying software quality prediction model. The underlying prediction model is a foundation upon which a module-order model is built, and due to this fact it is of great importance that an appropriate modeling technique be used to build the underlying prediction model. Module-order models use the prediction of the underlying model to rank modules with respect to their estimated number of faults. The purpose of this report is to present a comprehensive empirical study of some of the commonly used fault prediction techniques. Models build by these techniques can be used as underlying prediction models for the subsequent module-order models.

We evaluate the predictive performance of six commonly used fault prediction techniques: CART-LS (least squares algorithm), CART-LAD (least absolute deviation algorithm), S-PLUS, multiple linear regression, artificial neural networks, and case-based reasoning. Our research group at the Empirical Software Engineering Laboratory at Florida Atlantic University has performed experiments using all of the mentioned modeling techniques. The objective of our investigation is to determine the fault prediction performance order of the six techniques considered.

The case study presented in this report comprises of software metrics collected over four historical releases from a very large-scale telecommunications system.

Performance metrics, average absolute and average relative errors, are utilized to gauge accuracy of different prediction models. Comparative study is performed using models based on both, original software metrics (RAW) and their principle components (PCA) or domain metrics. Two-way ANOVA randomized-complete block design models with two blocking variables are designed with average absolute and average relative errors as response variables. System release and the model type (RAW or PCA) form the blocking variables and the prediction technique is treated as a factor. Using multiple-pairwise comparisons, the performance order of prediction models is determined.

We observe that for both average absolute and average relative errors, the CART-LAD model performs the best while the S-PLUS model is ranked sixth. The comparative study presented in this report can be used to build cost-effective module-order models that can facilitate in optimizing quality improvement resource usage.

Keywords: software quality prediction, software metrics, fault prediction, software reliability engineering, CART, S-PLUS, multiple linear regression, artificial neural networks, case based reasoning

1 Introduction

Software reliability is an important attribute of high-assurance and mission-critical systems. Such complex systems are heavily dependent on reliability and stability of their underlying software applications. The challenges involved in achieving high software reliability increases the importance in developing and quantifying measures for software quality. Early fault prediction [21], a proven technique in achieving high software reliability, can be used to direct cost-effective quality enhancement efforts to modules that are likely to have a high number of faults. A software fault is a defect that causes software failure in an executable product [29].

Previous research [20] has shown that software quality models based on software metrics [36, 37] can yield predictions with useful accuracy. Such models can be used to predict the response variable which can either be the class of a module (e.x., fault-prone or not fault-prone) or a quality factor (e.x., number of faults) for a module. The former is usually referred to as classification models [14, 23] while the latter is usually referred to as prediction models [9, 38, 41]. The focus of this paper is on the latter, i.e., prediction models. Software quality prediction models can predict quantities like number of faults and software development effort. Software metrics used by the model and the response variable are referred to as the independent variables and dependent variable respectively.

Over the last few decades many software quality modeling techniques have been developed and used in real life software quality predictions. A few commonly used modeling techniques for software quality estimation include, regression trees [9, 14, 23, 38, 41], artificial neural networks (ANN) [7, 24], case-based reasoning (CBR) [8, 23, 26], and multiple linear regression (MLR) [2]. Other recently developed techniques that have also been used include, fuzzy logic [42] and optimal set reduction [3]. Many of these techniques facilitate software quality estimation modeling using both classification and prediction models.

Despite the fact that currently many techniques are used to build and apply prediction models for real life software quality estimations, not many extensive studies have been done that compare the performance of commonly used prediction modeling techniques. Very few studies have performed comparative evaluations of a few of the available techniques and methods, for example, Finnie et al. [7] and Gray and Macdonnel [10].

Gray and Macdonnel [10], use three small-scale case studies to evaluate software development effort (or maintenance changes) estimation accuracy of prediction models built using MLR and ANN. The importance of factors other than predictive accuracy, such as data characteristics, expertise, and interpretability have been demonstrated, however, the comparative study lacks statistical verification [2]. The overall conclusion was that no single modeling technique can be used as a panacea for software effort estimation problems.

Finnie et al. [7] compare models built using ANN, CBR, and MLR. Similar to [7], this study compares software effort estimation accuracy of different modeling methods. It was concluded that both ANN and CBR gave similar accuracy, however, both of them yielded better results than MLR. Statistical verification using t-test was performed to determine the significance of their conclusions.

In both of the comparative studies mentioned, the case studies used were relatively small-scale and the research did not include other available prediction techniques, such as regression trees.

This study presents a comparative evaluation of predictive accuracy of six commonly used software quality prediction modeling techniques or algorithms. These are, CART-least squares (CART-LS), CART-least absolute deviation (CART-LAD), S-PLUS, MLR, ANN, and CBR. Software quality models that predict the number of faults in software modules, were built using all of the six techniques. Performance metrics, average absolute error (AAE) and average relative error (ARE) are used to gauge the fault prediction accuracy of modeling techniques. To our knowledge this is the first extensive study that compares the fault prediction capabilities of commonly used modeling techniques using a common large-scale case study.

The case study used to build models comprised of software metrics collected over four historical releases from a very large legacy telecommunications system, abbreviated as LLTS. Software metrics collected comprised of 24 product and 4 execution metrics, i.e., a total of 28 independent variables. Each system release has over 3500 updated modules or observations. A common model building and validation methodology was adopted for all six techniques. Release 1 was used to build the models while Release 2, 3 and 4 were used to validate the final models. For each of the modeling technique considered, models were built using both RAW and domain (PCA) metrics, and the models built are denoted as LLTS-RAW and LLTS-PCA models, respectively.

CART-LS, CART-LAD, and S-PLUS are tree-based prediction techniques [38] that provide simple white-box models which are attractive to analysts [4, 6]. To our knowledge these are the only tree-based prediction techniques currently available. CBR [23] is a problem solving technique which solves new problems by adapting solutions that were used to solve old problems [26]. ANN [24] adopt a learning approach to deriving a predictive model. MLR [2] is a traditional statistical means of predicting a dependent variable as a function of known independent variables. A more elaborate description of each of these methods is presented in the later sections.

Software metrics extracted (usually referred to as RAW metrics) from configuration and problem reporting systems are often heavily correlated to each other [22]. This is usually because they often represent measurements of related attributes of the given software system. The correlation among the independent variables can often lead to poor robustness and prediction accuracy of models built based on them. Principal components analysis (PCA) is a statistical technique that is used to alleviate the problems due to correlation of independent variables. As we will see shortly, we use domain (PCA) metrics in addition to RAW metrics to build and evaluate our prediction models.

The use of AAE and ARE for comparing different prediction techniques can sometimes be difficult and may lead to erroneous results. The problem is increased when comparing models based on multiple releases [20]. A scenario with two releases is presented that illustrates the possible difficulties. For Release 2, method A may have better AAE than method B, but for Release 3, method B may have better AAE than method A. The problem arises as to which release to use to compare methods A and B. The issue is further complicated with the use of over 2 releases to compare modeling methods.

The comparative approach adopted by Finnie et al. [7] is not suited for comparing models based on multiple releases. It is geared more towards case studies involving data collected from only one project release, i.e., *fit* and *test* data sets are extracted from the same release. A unique approach is adopted in our study to compare software quality estimation models. The approach compares the models over all system releases, which alleviates the problem addressed above.

Two-way ANOVA (analysis of variance) randomized-complete block design models with two blocking variables are built. These ANOVA models use AAE and ARE as the response variables. System release (Release 2, 3, and 4) and the model type (LLTS-RAW or LLTS-PCA model) form the two blocking variables, while the modeling technique or algorithm is treated as a factor. Release 1 is not used as a block since it was used to build or train the fault prediction models. Model type was used as a blocking variable (in ANOVA models) to observe if models built using RAW metrics were significantly apart from those built using domain metrics. It was observed that both models types performed similar, however, it should be noted that models based on principal components or domain metrics are more robust [22] than the corresponding models based on RAW metrics. ANOVA models indicated that both system releases and modeling techniques were significantly different from their respective counter parts. Multiple-pairwise comparisons [2] were performed to evaluate a performance or rank order of the six modeling techniques considered.

Comparisons of fault prediction accuracy (based on AAE and ARE) of the different modeling techniques considered in our study revealed the following performance order (decreasing accuracy): CART-LAD, CBR, MLR, ANN, CART-LS, and S-PLUS. It is indicated that CART-LAD and CBR have better fault prediction than MLR, ANN, CART-LS, and S-PLUS. The superior performance of CART-LAD and CBR as well as the inferior performance of S-PLUS is verified in a similar study that used data from other case studies [38].

The comparative technique presented in our study is not limited to only six modeling methods. It can be extended to compare fewer or more prediction modeling methods. However, data from multiple releases or multiple projects is needed to effectively utilize ANOVA design models (one-way or two-way) for performance comparisons of fault prediction models for software quality estimation.

The layout of the of rest the paper is as follows. Description of the different modeling methods is presented in section 2. In section 3, the adopted modeling methodology, comparative techniques used, and related technical concepts are presented. Section 4 describes the case study used in our study. Sections 5 and 6, present the results and conclusions of our comparative study.

2 Fault Prediction Techniques

This section presents a brief description of the fault prediction techniques compared in this paper. Our research group has performed extensive empirical research in software fault prediction modeling using all of the methods discussed, i.e, CART-LAD [38], CART-LS [38], S-PLUS [38], CBR [39], ANN [24, 39], and MLR [39].

2.1 Classification and Regression Trees

Classification and Regression Trees (CART) [4] is a statistical tool for tree structured data analysis. CART uses a regression tree to show how data may be predicted by a series of decisions at each internal node of the tree. In regression, a case consists of data $(\mathbf{x}_i, y_i)$ where, $\mathbf{x}_i$ is the i^{th} measurement vector of independent variables and y_i is the i^{th} response variable. The CART regression tree algorithm partitions the input data set into terminal nodes by a sequence of recursive binary splits. Binary splits are generated by CART, based on the significant independent variables. At each binary partition, the two subsets are as homogeneous as possible with respect to the dependent variable, which in our case is the number of faults in the software module.

CART regression tree algorithms initially build a large tree [4, 5], and then prune it backwards using *cross validation* to avoid overfitted trees [18]. Starting with the *fit* data set (learning sample L), three elements are necessary to determine a regression tree predictor:

1. A way to select a split at every internal node.

2. A rule for determining when a node is terminal.

3. A rule for assigning a value $\hat{y}(t)$ to every terminal node. $\hat{y}(t)$ is the predicted value of the response variable for terminal node t.

CART provides three ways to estimate the accuracy of regression trees; resubstitution estimate, test sample estimate and v-$fold$ cross validation estimate. Empirical studies in this paper use the v-$fold$ cross validation estimate to evaluate regression tree models. In a v-$fold$ cross validation estimate, the learning sample L (*fit* data set) is divided into v subsets of approximately equal size. (v-1) subsets are used as *fit* data sets while one remaining subset is used as a *test* data set. v such trials are performed such that each subset of the learning sample is used once as a *test* data set. The average error over these v trials, gives the cross validation error estimate. In our empirical studies we have used the 10-$fold$ cross validation estimate approach.

Certain parameters can be controlled when building regression trees with CART. These include number of terminal nodes, depth or level of regression tree, and node size before splitting. The first two parameters by default are automatically forecasted by the algorithm depending on the case study. The third parameter, node size, is set to 10 by default. In our experiments in regression tree modeling for the LLTS case study [38] we have used default values for these parameters.

In the following two sections we present the essential differences between the CART-LS (*Least Squares*) and CART-LAD (*Least Absolute Deviation*) methods of CART. Further in-depth mathematical details including, tree pruning methods and standard error estimates of the CART regression tree algorithms can be found in [4]. The final tree models are selected based on their cross validation[1] (mean square or average absolute deviation) relative errors, described in the following subsections.

2.1.1 Least Squares Method

This method generates regression trees using the within node mean value observed in each terminal node as its predicted value. Ranking of regression trees of different sizes is evaluated based on the *mean square error* estimate. Given a learning sample L consisting of $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_N, y_N)$, L is used to construct regression trees and also to estimate their accuracy.

The resubstitution error estimate for the CART-LS method, as a measure of accuracy for a regression tree T is given by,

$$R(T) = \frac{1}{N} \sum_{t \in T} \sum_{(\mathbf{x}_n, y_n) \in t} (y_n - \bar{y}(t))^2 \tag{1}$$

where, N is the total number of cases in the learning sample, t is a terminal node in the regression tree T, and $\bar{y}(t)$ is the mean value of response variables in t.

Given any set of possible splits $\mathbf{S}$ of a current terminal node t, the best split $\check{s}$ of t is that split in $\mathbf{S}$ which most decreases $R(T)$. For any split s of t into t_L and t_R, let $\Delta R(s,t) = R(t) - R(t_L) - R(t_R)$. Then the best split $\check{s}$ is given by,

$$\Delta R(\check{s}, t) = max[\Delta R(s,t)] \tag{2}$$

Thus, a regression tree is formed by iteratively splitting nodes so as to maximize the decrease in $R(T)$. The best split at a node is that split which most successfully separates the high response values from the low ones.

[1] Not to be confused with ARE (Section 3.2).

When using the test sample estimate, a *fit* data set is used to build regression trees, while a *test* data set is used to evaluate the accuracy of the tree models. The test sample error estimate for the CART-LS method is given by,

$$R^{ts}(T) = \frac{1}{N_2} \sum_{(\mathbf{x}_n, y_n) \in L_2} (y_n - d(\mathbf{x}_n))^2 \tag{3}$$

where, L_2 is the test data set with N_2 cases. $d(\mathbf{x}_n)$ denotes the predictor corresponding to the $\mathbf{x}_n$ measurement vector of independent variables. The learning sample L_1 is used to build regression trees, while the test sample L_2 is used to evaluate the accuracy of trees.

The v-$fold$ cross validation error estimate for the CART-LS method is given by,

$$R^{cv}(T) = \frac{1}{N} \sum_{v} \sum_{(\mathbf{x}_n, y_n) \in L_v} (y_n - d_v(\mathbf{x}_n))^2 \tag{4}$$

where, $d_v(\mathbf{x}_n)$ denotes the predictor for the v^{th} trial of the cross validation and L_v is the v^{th} subset of the learning sample L.

Let $\bar{y}$ be a sample mean of $y_1, ..., y_N$, and set $R(\bar{y})$ as,

$$R(\bar{y}) = \frac{1}{N} \sum_{n} (y_n - \bar{y})^2 \tag{5}$$

Then the mean square relative error estimates for resubstitution, test sample and v-$fold$ cross validation are given by, $R(T)/R(\bar{y})$, $R^{ts}(T)/R(\bar{y})$ and $R^{cv}(T)/R(\bar{y})$ respectively.

2.1.2 Least Absolute Deviation Method

This method generates regression trees using the within node median value observed in each terminal node as its predicted value. Ranking of regression trees of different sizes is evaluated based on the *mean absolute deviation* estimate.

The resubstitution error estimate $R(T)$ for the CART-LAD method is given by,

$$R(T) = \frac{1}{N} \sum_{t \in T} \sum_{(\mathbf{x}_n, y_n) \in t} |y_n - \tilde{y}(t)| \tag{6}$$

where, N is the total number of cases in the learning sample, t is a terminal node in the regression tree T and $\tilde{y}(t)$ is the median value of the y values in node t. The splitting criteria is analogous to that mentioned for the CART-LS method.

The test sample error estimate for the CART-LAD method is given by,

$$R^{ts}(T) = \frac{1}{N_2} \sum_{(\mathbf{x}_n, y_n) \in L_2} |y_n - d(\mathbf{x}_n)| \tag{7}$$

The v-$fold$ cross validation error estimate for the CART-LAD method is given by,

$$R^{cv}(T) = \frac{1}{N} \sum_{v} \sum_{(\mathbf{x}_n, y_n) \in L_v} |y_n - d_v(\mathbf{x}_n)| \tag{8}$$

Let $\tilde{y}$ be a sample median of $y_1, ..., y_N$, and set $R(\tilde{y})$ as,

$$R(\tilde{y}) = \frac{1}{N} \sum_{n} |y_n - \tilde{y}| \tag{9}$$

Then the absolute relative error estimates for resubstitution, test sample and v-$fold$ cross validation are given by, $R(T)/R(\tilde{y})$, $R^{ts}(T)/R(\tilde{y})$ and $R^{cv}(T)/R(\tilde{y})$ respectively.

2.2 S-PLUS Regression Trees

S-PLUS is a solution for advanced data analysis, data mining, and statistical modeling. It combines an intuitive graphical user interface with an extensive data analysis environment to offer ease of use and flexibility. Statistics in S-PLUS include regression tree models among other data mining functions. At the core of the S-PLUS system is S, the only language designed specifically for data visualization and exploration, statistical modeling and programming with data. *S* provides a rich, object-oriented environment designed for interactive data discovery. With a huge library of functions for all aspects of computing with data, *S* offers good extensibility.

In-depth mathematical details of the S-PLUS regression tree algorithm are presented in [6]. The predictors are software metrics treated by S-PLUS as ordinal measures, which are used to build regression trees to predict the response variable. The S-PLUS tree algorithm that can process only numerical data, constructs a regression tree which is a collection of decision rules. These rules are determined by recursive binary partitioning of the training data set.

Decision rules can be controlled by the analyst by specifying certain parameters [6], which limit the growth of the tree model. These parameters are *minsize*, the size threshold which limits the number of observations in a leaf node and *mindev*, the uniformity threshold which limits the allowable deviance in the leaf nodes. By controlling these parameters the analyst can prune the tree model to the desired level. However, S-PLUS provides a function that can be used to prune the tree after it has been constructed by the algorithm, without sacrificing the goodness-of-fit of the tree model [6]. In the course of the S-PLUS regression tree algorithm, modules in the *fit* data set are assigned to tree nodes. A software module is considered as an *object*.

Predictors are derived from software metrics as explained below. Let x_{ij} be the j^{th} predictor's value for module i, $\mathbf{x}_i$ be the vector of predictors for module i, and y_i be the response variable, i.e., number of faults. The algorithm initially assigns all the modules in the *fit* data set to the root node. The algorithm then recursively partitions each node's modules into two subsets that are assigned to its child nodes, until a stopping criterion halts further partitioning.

The deviance of module i is minus twice the log-likelihood, scaled by the variance, which reduces to the following [6].

$$D(\mu_i, y_i) = (y_i - \mu_i)^2 \tag{10}$$

where, μ_i is estimated by the mean value of y over all training modules that fall in the same leaf as module i. The deviance of a node l is the sum of the deviances of all the training modules in the node [6].

$$D(\mu_l; y) = \sum_{i \in l} (y_i - \mu_i)^2 \tag{11}$$

The tree-building algorithm chooses the predictor whose best split maximizes the change in deviance between the deviance of the current node and the sum of the deviances of the prospective child nodes. The "best split" of a predictor partitions the current node's set of modules into two subsets choosing the cutpoint that minimizes the sum of the deviances of the left and right prospective child nodes. Partitioning stops when the node deviance is less than a small fraction of the root node deviance.

$$\frac{D(\mu_l; y)}{D(\mu_{\mathrm{root}}; y)} < mindev \tag{12}$$

or the number of modules in the current node is less than a threshold,

$$n_l < minsize \tag{13}$$

where, $mindev$ and $minsize$ are parameters [38].

Let $L(\mathbf{x}_i)$ be the leaf that the i^{th} module falls into according to the structure of the tree. The predicted value of the response variable for module i is the mean of training modules in the leaf it falls into.

$$\hat{y}_i = \mu_{L(\mathbf{x}_i)} \tag{14}$$

Empirical studies using regression tree modeling with S-PLUS, was performed by our research team in [38]. Regression trees were built for the LLTS case study, by varying parameters *mindev* and *minsize*. For each of the models, performance metrics AAE and ARE was computed and the final model was selected based on quality of fit values [38].

2.3 Case Based Reasoning

A CBR [23, 39] system arrives at a solution by retrieving past instances of the same or a similar problem. The past instances are in a library of cases containing all known data. Each case in the library contains information about the program module it describes, which will include predictors and the response variable. A CBR system [26] can take advantage of availability of new or revised information, by adding new cases or by removing obsolete cases from the case library. Good scalability of CBR provides fast retrieval even as the size of the case library goes up. CBR systems can be designed to alert users when a new case is outside the bounds of current experience.

The first step is to determine a good software quality model that can predict the dependent variable with minimal error. This is done by varying the parameter which in this case is the value of *nearest neighbors*. Let N be the complete set of *nearest neighbors*, which are cases in the *fit* data set that are most similar to the present case in the target (*test*) data set. The number of *nearest neighbors*, n_N, (number of cases), is empirically determined by the user.

A CBR model is the training data with associated parameters like similarity functions, i.e., *Euclidean Distance*, *Absolute Difference*, or *Mahalonobis Distance*, solution algorithms, i.e., *Unweighted Average* or *Inverse Distance Weighted Average.* The case library is also known as the *fit* data set and the new cases, whose number of faults is to be predicted, is known as the *test* data set. The problem is to estimate the value of the dependent variable for a future or currently developed program module, early in its life cycle. The closer the predicted value is to the actual value, the better the accuracy of prediction.

The model selection is done by using the case library as both *fit* and target (*test*) data sets. Cross-validation is used to build the model. If the *fit* data set (case library) has n observations, at each iteration one case or observation is removed from the case library and the dependent variable (i.e., number of faults) is predicted using the remaining $n-1$ cases, i.e., the case library will have $n-1$ observations and the target (*test*) data set will have one observation. The one isolated observation acts as the *test* case to evaluate the prediction made by the $n-1$ cases. The prediction error (i.e., AAE and ARE) of the n iterations is computed and the CBR model with the least error is finally selected. The algorithms described below are used in the retrieval of the cases from the *fit* data set that are most similar to the target module and to estimate the dependent variable.

The RAW metrics in a software system module usually have vastly differing measurement units and highly varied ranges. Often, each metric has a unit of its own. Standardization is a technique that converts all the metrics to a uniform system of coordinates so that they will all have the same unit of measure. For each metric, x_i, the standardized metric is given by,

$$Z_i = \frac{x_i - \bar{x}_i}{s_i}$$

where, $\bar{x}_i$ is the mean and s_i is the standard deviation of the i^{th} metric, x_i. All independent variables in the data set are standardized to a mean of zero and a variance of one.

While other studies used normalization technique [5], we used standardization (except for *Mahalonobis Distance* function, in which neither is needed [39]).

A similarity function is used to compute the distance d_{ij} between the current module i and each of the modules j in the case library. Let c_{jk} be the value of the k^{th} independent variable of case j, and let $\mathbf{c}_j$ be the vector of independent variable values for case j. Let x_{ik} be the value of the k^{th} independent variable for target module i, and $\mathbf{x}_i$ be the vector of independent variable values for module i. The *Euclidean Distance* is given by,

$$d_{ij} = \left(\sum_{k=1}^{m} (w_k(c_{jk} - x_{ik}))^2 \right)^{\frac{1}{2}} \tag{15}$$

where, m is the number of independent variables and w_k is the weight of the k^{th} independent variable. The *Absolute Difference* or *City Block Distance* is given by,

$$d_{ij} = \sum_{k=1}^{m} w_k \left| c_{jk} - x_{ik} \right| \tag{16}$$

The *Mahalonobis Distance* is given by,

$$d_{ij} = (\mathbf{c}_j - \mathbf{x}_i)' \mathbf{S}^{-1} (\mathbf{c}_j - \mathbf{x}_i) \tag{17}$$

where, $(')$ means transpose, and $\mathbf{S}^{-1}$ is the inverse of the variance-covariance matrix of the independent variables for all the modules in the case library. $\mathbf{S}$ becomes an identity (unit) matrix and *Mahalonobis* becomes the *Euclidean* distance squared when the independent variables are orthogonal (zero correlation [22]) and have unit variance. When the independent variables are highly correlated and/or vary on vastly differing scales, the *Mahalonobis* distance is a very good alternative to other distance measures. Whenever the *Mahalonobis* measure is used, the independent variables do not need to be standardized or normalized.

The solution algorithm finally predicts the value of the dependent variable, y_i. The *Unweighted Average* solution algorithm is given by,

$$\hat{y}_i = \frac{1}{n_N} \sum_{j \in N} y_j \tag{18}$$

where, $\hat{y}_i$ is the mean value of the dependent variable of the most similar n_N modules from the case library.

The *Inverse Distance Weighted Average* solution algorithm is given by,

$$\delta_{ij} = \frac{1/d_{ij}}{\sum_{j \in N} 1/d_{ij}} \tag{19}$$

$$\hat{y}_i = \sum_{j \in N} \delta_{ij} y_j \tag{20}$$

where, y_i is estimated using the distance measures for the n_N closest cases as weights in a weighted average. Since smaller distances indicate a closer match and each case is weighted by a normalized inverse distance. The case most similar to the target module has the largest weight, thus playing a major role in prediction.

Fault prediction models were built by our research group [39, 16] for the LLTS case study. All three similarity functions mentioned above were considered. The number of similar cases, n_N, selected from the fit data set is a significant parameter during model building process. The model whose n_N results in the least value of AAE with cross validation is selected as the final model [39]. Experiments were therefore conducted by varying n_N, to find the optimum value. It was observed that the *Mahalonobis Distance* function gave better prediction results as compared to prediction obtained with *Euclidean Distance* and *Absolute Difference* functions. It was also observed that the *Inverse Distance Weighted Average* solution algorithm yielded better prediction than the *Unweighted Average* solution algorithm. Thus, *Inverse Weighted Distance Average* together with *Mahalonobis Distance* gave the best prediction results.

2.4 Artificial Neural Networks

Artificial neural networks (ANN) [2] are systems that are deliberately constructed to make use of some organizational principles resembling those of the human brain. ANN have been studied since Rosenblatt [34] first introduced single layer perceptrons. Because of the limitations of single-layer systems pointed out by Minsky and Papert [31], interest in ANN has been dwindling. Recent resurgence in the field of ANN was encouraged by the new learning algorithms [12], analog VLSI techniques, and parallel processing [28].

ANN can be classified, according to learning rules, into two categories, supervised-learning networks and unsupervised-learning networks [27]. The ANN we studied are supervised learning networks. In supervised-learning, at each instant of time when input is applied to an ANN, the corresponding desired response of the system is given. The network is thus told precisely what it should be emitting as output. In summary, we confine our study to feedforward supervised-learning neural networks, and backpropagation [28, 12] neural networks in particular. Figure 1 illustrates the structure of a feedforward supervised-learning neural network.

Neural networks consist of neurons. Figure 2 shows the structure of a neuron. In this model, the k^{th} processing element computes a weighted sum of its inputs x_j (independent variable) and basis b_k as the input to the activation function, the output of the activation function is the output of the neuron o_k (dependent variable). Suppose there are m inputs, $x_1, x_2, ..., x_m$, to the neurons and the weights associated with these inputs are

[2] Notations in this Section are independent to those of other sections. They are used exclusively for illustrating the theory of neural network.

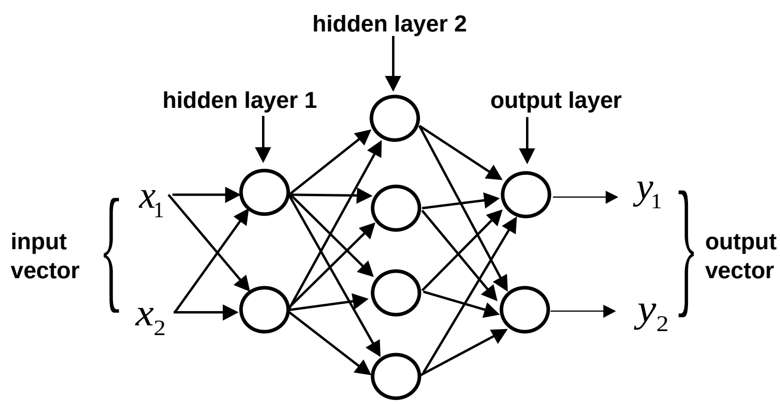

Figure 1: A feedforward neural network

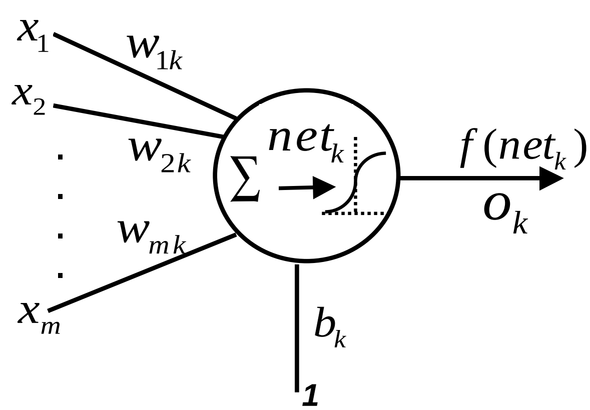

Figure 2: Anatomy of a neuron

$w_{1k}, w_{2k}, ..., w_{mk}$. So the operation of the neuron can be described as following.

$$net_k = w_{1k}x_1 + w_{2k}x_2 + ... + w_{mk}x_m + b_k \tag{21}$$

$$o_k = f(net_k) \tag{22}$$

where $f(\cdot)$ is the activation function of this neuron.

Backpropagation [35] is the most popular training algorithm for multilayer neural networks. The algorithm initializes the network with a random set of weights and basis, and the network trains from a set of input-output pairs. Each pair requires a two stage learning algorithm: *forward pass* and *backward pass*. The *forward pass* propagates the input vector through the network until it reaches the output layer. First the input vector propagates to the hidden units. Each hidden unit calculates the weighted sum of the input vector and its interconnection weights. Each hidden unit uses the weighted sum to calculate its activation. Next, hidden unit activations propagate to the output layer. Each node in the output layer calculates its weighted sum and activation. The output of the network is compared to the expected output of the input-output pair, and their difference (error vector) is used to train the network to minimize the error, this is called *backward pass*. First the error passes from the output layer to the hidden layer updating output weights. Next each hidden unit calculates an error based on the error from each output unit, the error from the hidden units updates input weight. The training stops only when the sum of squared error satisfies the requirement or the number of epochs passes the set point where an *epoch* means that all the training data go through the forward pass and backward pass once.

The least mean square algorithm computes the weight updates for each input sample and the weights are modified after each sample. This procedure is called *sample-by-sample learning*. An alternative solution is to compute the weight update for each input sample and store these values (without changing the weights) during one pass through the training set (epoch). At the end of the epoch, all the weight updates are added together, and only then will the weights be updated with the composite value. This is called *batch training* and is what we used in our case studies.

A neural network model was built for the LLTS case study (both RAW and PCA data sets) in [39]. Since the neural networks use the unipolar sigmoid function as their activation function for all the nodes, the dependent variable, number of faults, was scaled to the range [0,1]. After the training process, the result was converted back to the original scale. The training data set was normalized to avoid a slow network training process, and it was found that the training speed increased after normalization. The overall architecture of the final neural network model was determined empirically, and further details of our study is presented in [39].

2.5 Multiple Linear Regression

It is a statistical means of estimating or predicting a dependent variable as a function of known independent variables. It is an equation where the response variable is expressed in terms of predictors. The general form of a multiple linear regression (MLR) model can be given by

$$\hat{y}_i = a_0 + a_1 x_{i1} + \ldots + a_p x_{ip} \tag{23}$$

$$y_i = a_0 + a_1 x_{i1} + \ldots + a_p x_{ip} + e_i \tag{24}$$

where, $x_{i1}, \ldots, x_{ip}$ are the independent variables' values, $a_0, \ldots, a_p$ are the parameters to be estimated, $\hat{y}_i$ is the dependent variable to be predicted, y_i is the actual value of the dependent variable and $e_i = y_i - \hat{y}_i$ is the error in prediction for the i^{th} case.

The data available is initially subject to statistical analysis, with the aim to remove any correlation existing between independent variables and to remove insignificant independent variables, not accounting for the dependent variable. The process of determining the variables which are significant is known as *model selection*. Several methods of model selection exist. They are forward elimination, stepwise selection and backward elimination. Here, stepwise regression is used.

Stepwise regression [2] selects an optimal set of independent variables for the model.

In this process variables are either added or deleted from the regression model at each step of the model building process. Once the model is selected, the parameters $a_0, \ldots, a_p$ are then estimated using the *least squares* method. The values of the parameters are selected such that they minimize $\sum_{i=1}^{N} {e_i}^2$, where, N is the number of observations in the *fit* data set.

3 Methodology

In this section, we present a discussion of the approach adopted in comparing the different fault prediction modeling techniques discussed earlier. Theory and related principles of our comparative technique is also presented in this section.

3.1 Building Fault Prediction Models

The generic model building and validation approach adopted in fault prediction modeling with CART-LS [38], CART-LAD [38], S-PLUS [38], CBR [39], ANN [24], and MLR [20] is summarized in the following steps.

1. *Preprocessing Data*: A few modeling tools demand preprocessing of data before analysis. Some preprocessing may include, logarithmic transformation, standardization, and grouping of data.

2. *Formatting Data*: *fit* and *test* data sets may have to be converted to a format acceptable by the tool. For example, when using CART, data sets have to be converted to the SYSTAT file format [40].

3. *Building Prediction Models*: Release 1, the *fit* data set is used to build different models. Certain parameters specific to the modeling technique (section 2), are varied. Average absolute (AAE) and average relative errors (ARE) of models built are computed (for Release 1).

4. *Selecting Prediction Models*: Models with the lowest AAE and ARE values are selected as our final fault prediction models. In the case of CART-LS and CART-LAD the models were selected based on the lowest cross-validation relative error[3] value [38]. S-PLUS, ANN, and MLR models were selected based on their quality-of-fit values [18, 38]. A cross-validation (with lowest AAE) model selection approach [39] was adopted for CBR.

5. *Validating Prediction Models*: Release 2, 3, and 4 are used as *test* data sets to evaluate the prediction accuracy of the models selected. Performance metrics, AAE and ARE are computed. These are used to observe the estimation accuracy of models.

3.2 Performance Metrics

Fault prediction accuracy of the models selected is determined by estimating performance metrics. Two common statistics for evaluating predictions, average absolute error (AAE) and average relative error (ARE) are computed as,

$$AAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{25}$$

$$ARE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i + 1}\right| \tag{26}$$

[3]Not to be confused with ARE. Please refer to Section 2.1 for details.

where, n is the number of modules in the target data set. The denominator in ARE has a one added to avoid division by zero [25]. Our study compares fault prediction models using both AAE and ARE, since the effectiveness of one over the other is out of scope for this paper.

3.3 Analysis of Variance Models

ANOVA, abbreviated for Analysis of Variance, is a commonly used statistical technique when comparing differences between the means of three or more independent groups or populations. In our study we employ the two-way ANOVA: randomized complete block design modeling approach [2, 32], in which n *heterogeneous* subjects are classified into b homogeneous groups, called blocks so that the subjects in each block can then be randomly assigned, one each, to the levels of the factor of interest prior to the performance of a two-tailed F test, to determine the existence of significant factor effects.

Selecting the appropriate experimental design approach depends on the level of reduction in experimental error required. Since the primary objective for selecting a particular experimental design is to reduce experimental error (variability within data), a better design could be obtained if subject variability is separated from the experimental error [32]. A two-way ANOVA randomized complete block design, is a restricted randomization design in which the experimental units are first sorted into homogeneous groups, i.e., blocks, and the treatments are then assigned randomly within the blocks.

We are interested in observing if different prediction techniques are different from each other and if the system releases are different from each other. We are also interested in observing if principal components analysis [22] of the independent variables, results in better fault prediction accuracy of models. Substantial reduction in experimental errors can be obtained if more than one variable is used for determining blocks [2]. We designed two-way ANOVA models using two blocking variables, namely, system release and model type, i.e., models build based on RAW metrics and their principal components.

AAE and ARE values predicted by models for different releases and data sets (RAW and PCA) are the response variables in our experimental design models (ANOVA), which involve 6 factor treatments (six fault prediction techniques) and 2 blocking variables. The first one has 3 blocks (system releases 2, 3, and 4) while the second one has 2 blocks (RAW and PCA models). The p-values obtained from the ANOVA design models (Table 5), indicate the significance of the difference between the different modeling methods, between the different system releases, and between the models built using RAW and PCA metrics.

To develop the ANOVA procedure for a randomized complete block design, Y_{ijk}, the observation in the i^{th} block (i = 1, 2, ..., b) of B and the k^{th} block (k = 1, 2, ..., c) of C under the j^{th} factor level (j = 1, 2, ..., a), can be represented by the model,

$$Y_{ijk} = \mu + A_j + B_i + C_k + \epsilon_{ijk} \tag{27}$$

where,

μ = *overall* effect or mean common to all observations.

$A_j = \mu_{\cdot j \cdot}$ - μ, a treatment effect peculiar to the j^{th} level of factor A (method).

$B_i = \mu_{i \cdot\cdot}$ - μ, a block effect (system release) peculiar to the i^{th} block of B.

$C_k = \mu_{\cdot\cdot k}$ - μ, a block effect (model type, RAW or PCA) peculiar to the k^{th} block of C.

ϵ_{ijk} = *random variation* or *experimental error* associated with the observation in the i^{th} block of B and k^{th} block of C under the j^{th} level of factor A.

$\mu_{\cdot j \cdot}$ = true mean for the j^{th} level of factor A.

$\mu_{i \cdot \cdot}$ = true mean for the i^{th} block of B.

$\mu_{\cdot \cdot k}$ = true mean for the k^{th} block of C.

3.4 Hypothesis Testing: A *p*-value approach

Hypothesis testing is concerned with the testing of certain specified (i.e., *hypothesized*) values for those population parameters. Statisticians and software analysts alike, often perform hypothesis tests [2] when comparing different models. A *null hypothesis*, H_0, is tested against its compliment, the *alternate hypothesis*, H_A. Hypotheses are usually set up to determine if the data supports a belief as specified by H_A. These tests indicate the significance (α) of difference between two methods or populations.

The selection of the pre-determined significance level α, may depend on the analyst and the project involved. In some cases the selection of α, may be too ambiguous or difficult [1]. In such situations, it may be preferred to perform hypothesis testing without setting a value for α. This may be achieved by employing the p-value approach to hypothesis testing [1, 2]. This approach involves finding a value p, such that a given H_0, will not be accepted for any $\alpha \geq p$. Otherwise, H_0 will be not be rejected, i.e., $\alpha < p$. If this probability (p-value) is very high, H_0 is not rejected, while if this likelihood is very small (traditionally $\leq$ 0.05), H_0 is rejected. Hypotheses tests may be *one-tailed* or *two-tailed*, depending on the alternative hypothesis, H_A, of interest to the researcher [1, 2].

In this study, we use the *Minitab* software tool [1]. The tool has provision for statistical comparative analysis. We compute the p-values to determine if a prediction method is significantly better than another method. These p-values are used in deciding on the performance order of the different fault prediction methods. In making decisions regarding the rejection or non-rejection of H_0, the appropriate test statistic would be compared against the critical values for the particular sampling distribution of interest. For our comparative study, we use the F statistic [2]. If the test statistic F, is distributed as $F(\nu_1, \nu_2)$, then p-value is given by,

$$p = Pr\{F(p; \nu_1, \nu_2) \leq F(\nu_1, \nu_2)\} \tag{28}$$

where, ν_1 and ν_2 are the degrees of freedom for the F distribution, $F(p; \nu_1, \nu_2)$ is the entry in the F-table [1], and $F(\nu_1, \nu_2)$ is the computed statistic for the hypothesis test.

3.5 Multiple-Pairwise Comparison

The ANOVA block design models do not specify or indicate which means differ from which of the other means. Multiple comparison methods facilitate a more detailed information about the differences of these means. Specifically they provide a statistical technique to compare two methods (ex., method A and method B) at a time. A variety of multiple comparison methods are available, and for our study we employ Bonferroni's multiple comparison equation [1, 2]. Hypothesis testing using the p-value approach (section 3.4), is performed, yielding the p-value which indicates the level of difference between the two methods being compared. The null and alternate hypotheses used for the multiple-pairwise comparisons, using AAE are given by equations 29 and 30. Comparisons for ARE, are done by substituting ARE for AAE in the below stated equations.

$$H_0 : \text{AAE}_A \geq \text{AAE}_B \tag{29}$$

$$H_A : \text{AAE}_A < \text{AAE}_B \tag{30}$$

4 System Description

The data for the case study used in this paper was collected over four releases, from a very large legacy telecommunications system (abbreviated as LLTS). Each release has approximately 3500 to 4500 updated software modules. The software system is an embedded-computer application that included finite-state machines. The software was written in PROTEL, a high-level language, using the procedural development paradigm and was maintained by professional programmers in a large organization.

A software module was considered as a set of related source-code files. Fault data, collected at the module-level by the problem reporting system, comprised of faults discovered during post unit testing phases. Post unit testing phases recorded faults that were discovered before and after the product was released to customers. Faults that were discovered by clients were recorded only if the discovery resulted in changes to the source code of the module.

Configuration management data analysis, identified software modules that were unchanged from the prior release. Fault data collected from the problem reporting system were tabulated into problem reports and anomalies were resolved. The number of modules that had faults were too few to facilitate effective software quality modeling. As a result, we considered only the updated modules, i.e., those modules that were new or had at least one update to its source code since its prior release. For modeling, we selected updated modules with no missing data in relevant variables. These updated modules had several millions lines of code, and there were a few thousand of these modules in each release.

The set of available software metrics is usually determined by pragmatic considerations. A data mining approach is preferred in exploiting software metrics data [19], by which a broad set of metrics are analyzed rather than limiting data collection according to predetermined research questions. Data collection for LLTS involved extracting source code from the configuration management system. Measurements were recorded using the EMERALD software metrics analysis tool [11]. Preliminary data analysis selected metrics that were appropriate for our modeling purposes.

Software metrics for this system was collected over four different releases. We label these releases as Release 1, Release 2, Release 3, and Release 4. The number of observations in Release 1, Release 2, Release 3, and Release 4 were 3649, 3981, 3541, and 3978 respectively. The software metrics collected included 24 product metrics, 14 process metrics and 4 execution metrics. The 14 process metrics were not used in our empirical study, because our research study is focussed on early fault prediction of modules for software quality modeling. Only the software metrics used in our empirical study, are presented in this paper (tables 1 and 2). The data sets, consists of 28 independent variables that were used to predict the response variable, number of faults in a software module during post unit testing. We shall refer to this case study as LLTS-RAW.

The software product metrics in Table 1 are based on call graph, control flow graph, and statement metrics. An example of call graph metrics is number of distinct procedure calls. A module's control flow graph, consists of nodes and arcs depicting the flow of control of the program. Statement metrics are measurements of the program statements without implying the meaning or logistics of the statements. The problem reporting system maintained records on past problems. The proportion of installations that had

Table 1: LLTS Software Product Metrics

Symbol	Description
	Call Graph Metrics
$CALUNQ$	Number of distinct procedure calls to others.
$CAL2$	Number of second and following calls to others. $CAL2 = CAL - CALUNQ$ where CAL is the total number of calls.
	Control Flow Graph Metrics
$CNDNOT$	Number of arcs that are not conditional arcs.
$IFTH$	Number of non-loop conditional arcs, i.e., if-then constructs.
LOP	Number of loop constructs.
$CNDSPNSM$	Total span of branches of conditional arcs. The unit of measure is arcs.
$CNDSPNMX$	Maximum span of branches of conditional arcs.
$CTRNSTMX$	Maximum control structure nesting.
KNT	Number of knots. A "knot" in a control flow graph is where arcs cross due to a violation of structured programming principles.
$NDSINT$	Number of internal nodes (i.e., not an entry, exit, or pending node).
$NDSENT$	Number of entry nodes.
$NDSEXT$	Number of exit nodes.
$NDSPND$	Number of pending nodes, i.e., dead code segments.
$LGPATH$	Base 2 logarithm of the number of independent paths.
	Statement Metrics
$FILINCUQ$	Number of distinct include files.
LOC	Number of lines of code.
$STMCTL$	Number of control statements.
$STMDEC$	Number of declarative statements.
$STMEXE$	Number of executable statements.
$VARGLBUS$	Number of global variables used.
$VARSPNSM$	Total span of variables.
$VARSPNMX$	Maximum span of variables.
$VARUSDUQ$	Number of distinct variables used.
$VARUSD2$	Number of second and following uses of variables. $VARUSD2 = VARUSD - VARUSDUQ$ where $VARUSD$ is the total number of variable uses.

Table 2: LLTS Software Execution Metrics

Symbol	Description
USAGE	Deployment percentage of the module.
RESCPU	Execution time (microseconds) of an average transaction on a system serving consumers.
BUSCPU	Execution time (microseconds) of an average transaction on a system serving businesses.
TANCPU	Execution time (microseconds) of an average transaction on a tandem system.

a module, *USAGE*, was approximated by deployment data on a prior release [13]. Execution times in Table 2 were measured in a laboratory setting with different simulated workloads.

Software metrics extracted (usually referred to as RAW metrics) from configuration and problem reporting systems are often highly correlated to each other [22]. This is usually because they often represent measurements of related attributes of the given software system. The correlation among the independent variables can often lead to poor robustness and prediction accuracy of models built based on them. Principal components analysis (PCA) is a statistical technique that is used to alleviate the problems due to correlation of independent variables. Appendix A describes the details of principal components analysis.

The dimensionality of the 28 RAW metrics was reduced using PCA. Earlier research [20] indicated that the product and execution metrics groups of the RAW data were not correlated with each other. Hence, PCA was performed only on the 24 product metrics. Table 3 shows the six principal components extracted from the 24 product metrics of the LLTS-RAW data set. This table contains a 24×6 matrix, in which the 24 rows represent the product metrics while the 6 columns represent the principal components, *PROD1*,

Table 3: Factor Pattern of Product Metrics for LLTS-RAW

Metric	*PROD*1	*PROD*2	*PROD*3	*PROD*4	*PROD*5	*PROD*6
CALUNQ	**0.90241**	0.05180	0.10442	0.23226	0.17394	0.06161
VARUSDUQ	**0.89496**	0.18889	0.15268	0.17704	0.14681	0.19375
LOC	**0.88610**	0.28067	0.18160	0.16929	0.16431	0.14445
NDSENT	**0.87966**	-0.11142	0.01770	0.18394	0.10988	0.17201
STMEXE	**0.86869**	0.25870	0.17612	0.17324	0.26880	0.07169
STMCTL	**0.86701**	0.26070	0.27411	0.17258	0.08509	0.17429
NDSEXT	**0.84668**	0.01970	0.10855	0.20099	0.08568	0.35294
STMDEC	**0.84595**	0.20127	0.14148	0.14922	0.07117	0.14898
IFTH	**0.84569**	0.34158	0.27880	0.18162	0.10404	0.10659
NDSINT	**0.84185**	0.34355	0.27606	0.15248	0.18487	0.10920
CNDNOT	**0.83478**	0.31173	0.26233	0.15217	0.23697	0.17495
LOP	**0.82816**	0.10817	0.20842	0.01714	0.02129	-0.09590
VARGLBUS	**0.80191**	0.35962	0.20123	0.14369	0.21197	0.20453
VARUSD2	**0.79088**	0.44096	0.27108	0.11186	0.18082	0.12928
CAL2	**0.59715**	0.20418	0.07284	0.19317	0.56903	-0.05255
VARSPNSM	0.39174	**0.86022**	0.17718	0.10430	0.06747	0.08423
VARSPNMX	0.14039	**0.83489**	0.17722	0.35150	0.10357	0.09136
CNDSPNMX	0.12121	0.27629	**0.75661**	0.14289	0.25648	0.30600
CTRNSTMX	0.32233	0.09595	**0.70922**	0.42101	-0.00726	-0.01574
CNDSPNSM	0.60974	0.21553	**0.64240**	0.00704	0.22007	0.13087
FILINCUQ	0.39561	0.25790	0.15541	**0.72651**	-0.03570	0.16963
LGPATH	0.21017	0.37957	0.35793	**0.63962**	0.16986	-0.04151
KNT	0.21362	0.06906	0.17464	-0.00640	**0.88896**	0.09719
NDSPND	0.40212	0.14886	0.21690	0.07507	0.08412	**0.81557**
Variance	11.61638	2.82091	2.37167	1.69515	1.64281	1.23002
% Var.	48.40%	11.75%	9.88%	7.06%	6.85%	5.13%
Cum. %	48.40%	60.15%	70.03%	77.09%	83.94%	89.07%

Stopping rule: at least 89% of variance

PROD2, *PROD3*, *PROD4*, *PROD5* and *PROD6*. Each element in the matrix indicates the correlation between a principal component and a raw metric. These 6 principal components and the 4 execution metrics in Table 2 form the second set of data sets. We refer to this case study as LLTS-PCA.

Table 4: LLTS-RAW: AAE and ARE values

Model based on LLTS-RAW case study						
Modeling Method	*Release*2		*Release*3		*Release*4	
	AAE	ARE	AAE	ARE	AAE	ARE
CART-LS	0.948	0.618	0.942	0.602	1.407	0.838
CART-LAD	0.705	0.324	0.803	0.391	0.867	0.418
S-PLUS	0.909	0.577	0.954	0.602	1.267	0.774
CBR	0.884	0.585	0.861	0.499	0.831	0.492
ANN	0.946	0.584	1.016	0.620	1.249	0.749
MLR	0.890	0.571	0.960	0.602	0.926	0.584

Model based on LLTS-PCA case study						
Modeling Method	*Release*2		*Release*3		*Release*4	
	AAE	ARE	AAE	ARE	AAE	ARE
CART-LS	0.972	0.647	0.975	0.633	1.113	0.682
CART-LAD	0.727	0.344	0.823	0.407	0.860	0.456
S-PLUS	0.925	0.602	0.973	0.621	1.568	0.948
CBR	0.835	0.523	0.871	0.519	0.810	0.477
ANN	0.887	0.555	0.948	0.576	0.989	0.615
MLR	0.875	0.567	0.976	0.626	0.954	0.637

5 Empirical Results

Two models were built for the LLTS case study using each of the six prediction techniques, namely, CART-LS, CART-LAD, S-PLUS, CBR, ANN, and MLR. The first model was built using the LLTS-RAW data set while second model was built using the LLTS-PCA data set. Performance metrics AAE and ARE are computed for the both models, and are shown in Table 4. The values shown in Table 4 are for Release 2, 3, and 4 only since Release 1 was used as the *fit* data set.

The two-way ANOVA randomized complete block design models built using AAE and ARE as response variables, comprised of two blocking variables, i.e., system release and model type (RAW and PCA) and one factor, i.e., prediction technique. ANOVA models

Table 5: ANOVA models for LLTS case study

Source	DF	SS	MS	F	p-value
	Average Absolute Error				
Technique	5	0.4262	0.0852	5.99	0.001
Model Type	1	0.0022	0.0022	0.16	0.695
Release	2	0.2460	0.1230	8.64	0.001
Error	27	0.3842	0.0142		
Total	35	1.0586			
	Average Relative Error				
Technique	5	0.3688	0.0738	15.8	0.000
Model Type	1	0.0000	0.0000	0.00	0.989
Release	2	0.0657	0.0329	7.02	0.004
Error	27	0.1264	0.0047		
Total	35	0.5609			

Table 6: Multiple Pairwise Comparisons: p-values

Average Absolute Error						
	CART-LAD	CART-LS	S-PLUS	ANN	MLR	CBR
CART-LAD	*	0.0019	0.0004	0.0134	0.1515	0.7328
CART-LS	1.0000	*	0.8092	0.9995	1.0000	1.0000
S-PLUS	1.0000	0.9982	*	1.0000	1.0000	1.0000
ANN	1.0000	0.7158	0.3850	*	0.9999	1.0000
MLR	1.0000	0.1670	0.0506	0.5318	*	1.0000
CBR	0.9993	0.0124	0.0028	0.0742	0.4826	*
Average Relative Error						
	CART-LAD	CART-LS	S-PLUS	ANN	MLR	CBR
CART-LAD	*	0.0000	0.0000	0.0000	0.0000	0.0090
CART-LS	1.0000	*	0.8682	1.0000	1.0000	1.0000
S-PLUS	1.0000	0.9959	*	1.0000	1.0000	1.0000
ANN	1.0000	0.3877	0.1945	*	0.9966	1.0000
MLR	1.0000	0.1821	0.0773	0.8548	*	1.0000
CBR	1.0000	0.0014	0.0004	0.0414	0.1134	*

Table 7: Performance Order: LLTS case study

Average Absolute Error										
CART-LAD	<	CBR	<	MLR	<	ANN	<	CART-LS	<	S-PLUS
Average Relative Error										
CART-LAD	<	CBR	<	MLR	<	ANN	<	CART-LS	<	S-PLUS

were built over all the test data sets, i.e., Release 2, 3, and 4. The results of the ANOVA models are presented in Table 5. Notations of Table 5 are, DF - degrees of freedom, SS - sums of squares, MS - mean squares, and F - the F statistic [2].

It is observed from Table 5, for both AAE and ARE the system releases are significantly apart from each other (p-value = 0.001 and 0.004, respectively). It is also seen that the prediction techniques are also significantly apart from each other, i.e, p-value = 0.001 (AAE) and 0.000 (ARE). However, the LLTS-RAW and LLTS-PCA models interestingly performed similar, i.e., have similar prediction accuracies. Since the AAE and ARE values of the prediction techniques are significantly different from each other, we proceeded with multiple-pairwise comparisons of the different techniques.

Each of the six modeling methods is compared with the other five methods using a one-tailed pairwise comparison. For example, CART-LAD is compared with CART-LS, S-PLUS, CBR, ANN, and MLR individually. Thus for each pair of techniques we have two comparisons, for example, is CART-LAD better than CBR? and is CBR better than CART-LAD?. The p-value of the comparison is computed, and is used to observe whether a method is better than the one it is compared with.

Table 6 presents the p-values obtained from multiple pairwise comparisons. Comparisons are shown for both AAE and ARE performance metrics. The table can be viewed as a matrix, i.e., each pair of two methods forms a comparison. This implies each method listed in the first column is compared with (except itself) methods listed as headings of subsequent columns. For example, CART-LAD vs. CART-LS, CART-LAD vs. S-PLUS, and so on. Methods are not compared to themselves, and this is indicated by a '*' in the two tables. Since we compared six prediction techniques, there are 30 comparisons for each of the performance metrics. The p-values indicate the significance level of the difference in AAE or ARE values between two prediction techniques, and are used to conclude the final performance or rank order.

To indicate how we inferred the final performance order of the prediction methods, we present details for ARE in the next few paragraphs. A similar approach was followed in computing the performance order for AAE. Lets look at the ARE section of Table 6. Comparisons CART-LAD vs CART-LS, CART-LAD vs S-PLUS, CART-LAD vs ANN, CART-LAD vs MLR, and CART-LAD vs CBR have very low p-values, therefore indicate that CART-LAD has better predictive accuracy than the other five techniques. Hence, CART-LAD will be ranked first in the final order. Lets denote this deduction as **Da**.

From comparisons CBR vs CART-LAD ($p = 1.0000$), CBR vs CART-LS ($p = 0.0014$), CBR vs S-PLUS ($p = 0.0004$), CBR vs ANN ($p = 0.0414$), and CBR vs MLR ($p = 0.1134$) it is observed that CBR is better than all techniques except CART-LAD (verified by **Da**). Hence, CBR will be ranked second in the final order. Lets denote this deduction as **Db**.

Comparisons MLR vs CART-LS ($p = 0.1821$), MLR vs S-PLUS ($p = 0.0773$), MLR vs ANN ($p = 0.8548$), and ANN vs MLR ($p = 0.9966$) indicate that MLR is significantly better than both CART-LS and S-PLUS, but is only slightly better than ANN. Using this observation together with **Da** and **Db**, we conclude that MLR will be ranked third in the final order.

From comparisons ANN vs CART-LS ($p = 0.3877$) and ANN vs S-PLUS ($p = 0.1945$) we observe that ANN is better than both CART-LS and S-PLUS, and will be placed fourth in the rank order since we already have the first three. CART-LS will be placed before S-PLUS because comparisons CART-LS vs S-PLUS ($p = 0.8682$) and S-PLUS vs CART-LS ($p = 0.9959$) demonstrate that CART-LS performs slightly better than S-PLUS. Hence, CART-LS and S-PLUS will be placed fifth and sixth in the final rank order.

Performance orders for both AAE and ARE are shown in Table 7. The modeling techniques are ordered from left to right with decreasing prediction accuracy. The symbol '$<$' in the table indicates that the left hand side method has better fault prediction than the method on the right hand side. Thus, it is observed that CART-LAD and CBR yield better fault prediction as compared to MLR and ANN, which in turn are better predictors than CART-LS and S-PLUS.

6 Conclusions and Future Work

Software reliability is an important attribute of high-assurance and mission-critical systems. Such complex systems are heavily dependent on reliability and stability of their underlying software applications. The challenges involved in achieving high software reliability increases the importance in developing and quantifying measures for software quality. Early fault prediction, a proven technique in achieving high software reliability, can be used to direct cost-effective quality enhancement efforts to modules that are likely to have a high number of faults.

Software quality models based on software metrics can yield predictions with useful accuracy. Such models can be used for early fault predictions in software quality estimation applications. Fault prediction models, based on software metrics, can predict the number of faults in software modules.

In this paper we compare the fault prediction accuracies of six commonly used prediction modeling techniques, CART-LS, CART-LAD, S-PLUS, CBR, ANN, and MLR. The large-scale case study used in this comparative study, comprised of data collected over four historical system releases of a very large legacy telecommunications system. Models were built using RAW metrics as well as domain metrics (PCA). Two-way ANOVA models, with two blocking variables (system release and model type) were designed (over all releases) to investigate, if the releases were different from each other, if the techniques were different from each other, and if the RAW models were different from the corresponding PCA models. The ANOVA models were designed with average absolute error and average relative error as the response variables.

From the ANOVA models, it was observed that the releases and the modeling methods were significantly different than their respective counterparts, while the RAW models and PCA models gave similar results. Therefore, it is indicated that PCA may not necessarily improve fault prediction accuracy of software quality models. However, it should be noted that PCA removes correlation among the RAW metrics and the resulting models are more robust.

Multiple-pairwise comparisons for the six modeling techniques were performed, and a performance or rank order was determined based on the p-values obtained. The comparisons were performed for both AAE and ARE. The rank order of the six modeling methods suggest that CART-LAD and CBR have superior fault prediction accuracy than MLR, ANN, CART-LS, and S-PLUS. In the final rank order for both AAE and ARE, CART-LAD was ranked first while S-PLUS was ranked sixth.

Future work in related research areas may include investigating a similar comparative study, with software metrics from a software system other than a telecommunications system.

Acknowledgments

We thank Kenneth McGill for his encouragement and support, and Dr. Bojan Cukic for his helpful suggestions. We also thank John P. Hudepohl, Wendell D. Jones and the EMERALD team for collecting the case-study data. This work was supported in part by Cooperative Agreement NCC 2-1141 from NASA Ames Research Center, Software Technology Division, and Center Software Initiative for the NASA software Independent Verification and Validation Facility at Fairmont, West Virginia. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor.

References

[1] G. P. Beaumont. *Statistical Tests: An Introduction with Minitab Commentary.* Prentice Hall, 1996.

[2] M. L. Berenson, D. M. Levine, M. Goldstein. *Intermediate Statistical Methods And Applications.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1983, ISBN 0-13-470781-8.

[3] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044, Nov. 1993.

[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification And Regression Trees*. Wadsworth International Group, Belmont, California, 2nd. edition, 1984.

[5] L. C. Briand, Tristen Langley, and Isabella Wieczorek. A replicated assessment and comparison of common software cost modeling techniques. *International Conference on Software Engineering*, ACM, pages 113-206, Limerick Ireland, Sept. 2000.

[6] L. A. Clark and D. Pregibon. Tree-based models. In J. M. Chambers and T. J. Hastie, editors, *Statistical Models in S*, pages 377–419. Wadsworth, Pacific Grove, California, 1992.

[7] G. R. Finnie, G. E. Wittig, and J. M. Desharnais A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models. *Journal of Systems and Software*, 39:281–289, 1997.

[8] K. Ganesan, T. M. Khoshgofaar, and E. B. Allen. Case-based software quality prediction. *International Journal of Software Engineering and Knowledge Engineering*, 10(2):139–152, 2000.

[9] S. S. Gokhale and M. R. Lyu. Regression tree modeling for the prediction of software quality. In H. Pham, editor, *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*, pages 31–36, Anaheim, CA, Mar. 1997. International Society of Science and Applied Technologies.

[10] A. R. Gray and S. G. Macdonnel Software metrics data analysis - Exploring the relative performance of some commonly used modeling techniques. *Journal of Empirical Software Engineering*, 4:297–316, 1999.

[11] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. EMERALD: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, Sept. 1996.

[12] R. Hecht-Nielson. Applications of counter propogation network. *Neural Networks*, 1:131–139, 1988.

[13] W. D. Jones, J. P. Hudepohl, T. M. Khoshgoftaar, and E. B. Allen. Application of a usage profile in software quality models. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 148–157, Amsterdam, Netherlands, Mar. 1999. IEEE Computer Society.

[14] T. M. Khoshgoftaar and E. B. Allen. Modeling software quality with classification trees. In *Recent Advances in Reliability and Quality Engineering*, Hoang Pham Editor, World Scientific, Ch. 15, 247–270, 2001.

[15] T. M. Khoshgoftaar and E. B. Allen. Modeling the risk of software faults. Technical Report TR-CSE-00-06 (37 pages), Feb. 2000. Florida Atlantic University, Boca Raton FL USA. Submitted to NASA Independent Verification and Validation Facility, West Virginia.

[16] T. M. Khoshgoftaar, E. B. Allen, and J. C. Busboom. Modeling software quality: The software measurement analysis and reliability toolkit. In *Twelfth International Conference on Tools with Artificial Intelligence*, pages 54–61, Vancouver, Canada, Nov. 2000. *IEEE* Computer Society.

[17] T. M. Khoshgoftaar, E. B. Allen, and B. Cukic. Achieving high software reliability. *FY00*: Cooperative Agreement NCC 2-1411, NASA Ames Research Center, and *FY01 Center Initiative Proposal*: NASA Independent Verification and Validation Facility, West Virginia, June 2000.

[18] T. M. Khoshgoftaar, E. B. Allen, and J. Deng. Controlling overfitting in software quality models: Experiments with regression trees and classification. In *Proceedings of the Seventh International Software Metrics Symposium*, pages 190–198, London, England UK, April 2001. IEEE Computer Society.

[19] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–563, 1999.

[20] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 9:103–116, 2000.

[21] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, Jan. 1996.

[22] T. M. Khoshgoftaar, E. B. Allen, and R. Shan. Improving tree-based models of software quality with principal components analysis. In *Proceedings of the Eleventh International Symposium on Software Reliability Engineering*, pages 198–209, San Jose, California USA, October 2000. IEEE Computer Society.

[23] T. M. Khoshgoftaar, K. Ganesan, E. B. Allen, and F. D. Ross. Predicting fault-prone modules with case-based reasoning. In *The Eighth International Symposium on Software Reliability Engineering*, pages 27–35, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

[24] T. M. Khoshgoftaar and D. L. Lanning. A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software*, 29(1):85–91, Apr. 1995.

[25] T. M. Khoshgoftaar, J. C. Munson, B. B. Bhattacharya, and G. D. Richardson. Predictive modeling techniques of software quality from software measures. *IEEE Transactions on Software Engineering*, 18(11):979–987, Nov. 1992.

[26] J. L. Kolodner. *Case-Based Reasoning*. Morgan Kaugmann Publishers, San Mateo, CA, 1993.

[27] C. T. Lin and C. S. G. Lee. *Neural Fuzzy Systems: A Neuro-Fuzzy Synergism to Intelligent Systems*. Prentice Hall, Inc., Upper Saddle River, New Jersey 1966.

[28] R. P. Lippmann. An introduction to computing with neural networks. *IEEE Acoustics Speech, and Signal Processing Magazine*, 4(2):4–22, 1987.

[29] M. R. Lyu. Introduction. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 1, pages 3-25. McGraw-Hill, New York, 1996.

[30] J. C. Munson and T. M. Khoshgoftaar. The dimensionality of program complexity. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 245–253, Pittsburgh, Pennsylvania USA, May 1989. IEEE Computer Society. IEEE Computer Society.

[31] M. Minsky and S. Papert. *Perceptrons* MIT Press, MA, 1969

[32] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. In *Applied Linear Statistical Models*. Tom Casson, 1996.

[33] N. Ohlsson, M. Zhao, and M. Helander. Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 7:51–66, 1998.

[34] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, New York, 1962.

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel Distributed Processing*, volume 1, ch. 8, MIT Press, Cambridge, MA, 1986.

[36] N. F. Schneidewind. Software metrics validation: Space Shuttle flight software example. *Annals of Software Engineering*, 1:287–309, 1995.

[37] N. F. Schneidewind. Software metrics model for integrating quality control and prediction. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 402–415, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

[38] N. Seliya. Software fault prediction using tree based models. Master's thesis, Florida Atlantic University, Boca Raton, Florida USA, August 2001. Advised by Taghi M. Khoshgoftaar.

[39] N. Sundaresh. An empirical study of analogy based software fault prediction. Master's thesis, Florida Atlantic University, Boca Raton, Florida USA, May 2001. Advised by T. M. Khoshgoftaar.

[40] D. Steinberg and P. Colla. *CART: A supplementary module for SYSTAT.* Salford Systems, San Diego, CA, 1995.

[41] J. Troster and J. Tian. Measurement and defect modeling for a legacy software system. *Annals of Software Engineering*, 1:95–118, 1995.

[42] Z. Xu. Fuzzy logic techniques for software reliability engineering. Ph.D. dissertation, Florida Atlantic University, Boca Raton, Florida USA, May 2001. Advised by T. M. Khoshgoftaar.

A Principal Components Analysis

Software metrics extracted (RAW metrics) from configuration and problem reporting systems are often heavily correlated to each other [22]. This is usually because they often represent measurements of related attributes of the given software system. The correlation among the independent variables, can often lead to poor robustness and prediction accuracy of models built based on them. Principal components analysis (PCA) is a statistical technique that is used to alleviate the problems due to correlation of independent variables.

The RAW metrics are transformed into a smaller set of linear combinations that account for, most if not all the variance of the RAW data set. PCA also reduces the number of independent variables used in building models. The principal component variables are called domain metrics as compared to original independent variables which form the RAW metrics. The first principal component, accounts for the largest fraction of the total variance in the original data. Let's denote the first component by PC_1. Thus PC_1 is the linear combination of the observed independent variables x_j, *where*

$j = 1, 2, \ldots, m$.

$$PC_1 = w_{(1)1}x_1 + w_{(1)2}x_2 + \cdots + w_{(1)m}x_m \tag{31}$$

In the above equation, the weights $w_{(1)1}, w_{(1)2}, \ldots, w_{(1)m}$ have been chosen to maximize the ratio of the variance of PC_1 to the total variance, subject to the constraint that $\sum_{j=1}^{m} w_{(1)j}^2 = 1$. The second principal component, PC_2, is the weighted linear combination of the observed variables that are not correlated with the first linear combination (i.e., PC_1), and accounts for the maximum amount of the remaining total variance. Let's denote the second component by PC_2. In general, the i^{th} principal component is the weighted linear combination of the x's and is given by,

$$PC_i = w_{(i)1}x_1 + w_{(i)2}x_2 + \cdots + w_{(i)m}x_m \tag{32}$$

It is possible to extract the same number of principal components as the number of the original variables. The goal however, is to account for most of the total variance with as few principal components as possible. Therefore, a stopping rule is introduced to choose as few domain metrics as possible. Hence, given m software metrics, a stopping rule chooses $p \ll m$ domain metrics and ignores the remaining domain metrics because they have insignificant variation across the data set. The stopping rule, terminates principal components analysis, once a particular variance is accounted for during analysis.

Suppose we have m product measurements on each of the n modules. PCA performs the following calculations, given an $n \times m$ matrix of standardized metric data, $\mathbf{Z}$.

1. Compute the *covariance matrix*, $\mathbf{\Sigma}$, of $\mathbf{Z}$.

2. Compute the *eigenvalues*, λ_j, and the *eigenvectors*, $\mathbf{e}_j$, of $\mathbf{\Sigma}$, $j = 1, \ldots, m$.

3. Minimize the dimensionality of the data. If we choose to explain at least 90% of the total variance of the original standardized metrics, we then choose the minimum p such that $\sum_{j=1}^{p} \lambda_j / m \geq 0.90$.

4. Compute a standardized transformation matrix $\mathbf{T}$, where each column is defined as,

$$\mathbf{t}_j = \frac{\mathbf{e}_j}{\sqrt{\lambda_j}} \text{ for } j = 1, \ldots, p \tag{33}$$

5. Compute the domain metrics for each module, where

$$D_j = \mathbf{Z}\mathbf{t}_j \tag{34}$$

$$\mathbf{D} = \mathbf{Z}\mathbf{T} \tag{35}$$

The final result, of a principal components analysis of a given raw metrics, is an $n \times p$ matrix of domain metrics data $\mathbf{D}$, where each domain metric, D_j, has a mean of zero and a unit variance.